Pergamon

Appl. Math. Lett. Vol. 8, No. 5, pp. 11–15, 1995

0893-9659/95 $9.50 + 0.00

0893-9659(95)00059-3

Sparse Logical Terms

A. FALL
School of Computing Science, Simon Fraser University
Burnaby, B.C. V5A-1S6, Canada
fall@cs.sfu.ca

(Received May 1995; accepted June 1995)

Abstract—We propose a sparse representation for logical terms analogous to formalisms used for sparse matrices. For applications which manage terms containing many anonymous variables, this can provide a savings both in terms of storage space and unification time. Variations of this scheme provide a set of easily implemented tools suitable for diverse applications such as taxonomic encoding, natural language processing, and automatic configuration.

Keywords—Herbrand terms, Logic programming, Prolog, Sparse representations.

1. INTRODUCTION

Compact representations for data structures are commonly used when certain properties can be exploited to significantly reduce the storage space required. As an example, principles of locality are used in data compression techniques. For sparse matrices, the assumption that the majority of elements are zero permits us to retain only the nonzero elements, along with their coordinates. If this assumption holds true, the savings accrued by not explicitly storing the zero elements outweighs the additional cost of storing coordinates for nonzero entries.

We propose a similar representation for logical terms, as used in Prolog. A sparse term is a term in which the majority of elements (i.e., functors, atoms and variables) are anonymous variables. Named variables provide coreference between term positions, whereas the only purpose of anonymous variables is to reserve positions, and so they do not contribute to the information content of a term. In Prolog, an anonymous variable is represented by an underscore.

Applications which work with sparse terms can benefit from our proposal both in terms of space and time. Unification with an occurs check needs only to examine the named variables. Unification without an occurs check is linear in the sum of the number of atoms, functors and variables of the two terms. This will be more efficient as our sparse representation eliminates the storage of anonymous variables.

In this paper, we outline a sparse representation for logical terms and discuss when it may be beneficial. Then we describe variations to our scheme which will permit more flexible use of logical terms as well as the incorporation of uncertainty beyond that offered by variables.

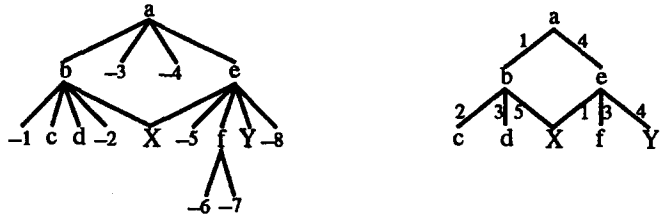
2. REPRESENTING SPARSE TERMS

Our representation is modeled after that of sparse matrices. An $n \times m$ sparse matrix may be stored as a list of coordinate/value pairs for the nonzero elements rather than as an $n \times m$ array. For example, the following matrix can be stored as [(1,2)-1, (2,4)-5, (4,2)-3, (4,5)-4]:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 4 \end{bmatrix}.$$

We avoid storing the zeros by using a more space-consuming representation for the nonzero elements. By assuming that most of the elements are zeros, we predict a net reduction in storage space.

A sparse term representation will relieve us from storing anonymous variables at the expense of a more complex scheme for the named elements (i.e., atoms, functors and named variables). We focus on the surface form of terms. Although the internal representation may be quite different from this and is implementation dependent, it is the surface form that users manipulate and store outside the system. As for sparse matrices, we need to store the position, or index, of the named elements. Using a rooted graph notation, we can do this by labeling arcs with the index of the named elements and removing the anonymous variables. Consider, for example, the term $a(b(\__1,c,d,\__2,X),\__3,\__4,e(X,\__5,f(\__6,\__7),Y,\__8))$[1]. Both the ordinary and sparse forms of this term are shown below in a rooted graph notation:



The sparse term can be represented linearly as: a-[1.b-[2.c, 3.d, 5.X], 4.e-[1.X, 3.f, 4.Y]], where the lists are ordered according to increasing index. To be more precise, we provide the following definition of our representation.

DEFINITION 2.1. *A sparse term is either (i) an atom (ii) a named variable or (iii) a functor of the form a-L, where a is the functor symbol and L is a sparse argument list. A sparse argument list is a list of elements of the form n.ST, where ST is a sparse term and n is the index of ST in the parent term. This list is ordered by increasing indices with no repetitions.*

2.1. Space Requirements

Now that we have a sparse representation for logical terms, when is a term considered sparse? That is, when will this representation benefit an application? Since an accurate account of the space required to represent a logical term, e.g., in Prolog, is implementation dependent, we will restrict our analysis to the asymptotic time and space behavior of the surface form.

Consider an ordinary term which has n named elements and m anonymous variables. Since there are $n+m$ symbols, let us assume representing each requires $O(\log(n+m))$ space. For the sparse representation, only $O(\log n)$ space is required. Both representations will require space for the n named elements, and since they are both logarithmic, we do not include this factor in our calculations. For punctuation marks (e.g., commas, parentheses, dashes), ordinary terms require $O(n+m)$ space whereas sparse terms require $O(n)$ space. Since punctuation may not form part of the internal representation, we do not consider it further.

In addition to the above, ordinary terms require $O(m\log(n+m))$ space for anonymous variables, whereas sparse terms require $O(n\log(n+m))$ space for indices. Essentially, this means that the space benefits of our sparse representation begin to manifest when the ratio of anonymous variables to named elements is greater than one. Of course, due to the constants not included in this analysis, these benefits may not become evident until this ratio is somewhat greater than this.

[1]The anonymous variables have been subscripted for clarity.

2.2. Unification

Without an occurs check, unification of both ordinary and sparse terms is linear in the number of symbols involved. If the number of named elements in both terms is n and the number of anonymous variables is m, we have $O(n + m)$ for ordinary terms vs. $O(n)$ for sparse terms. For unification with an occurs check, we avoid needlessly checking the anonymous variables. In both cases, we achieve asymptotically better results. Thus, by using our sparse representation, applications involving sparse terms have potential benefit both in terms of time and space.

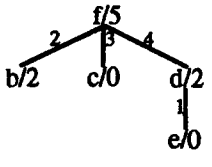
3. VARIATIONS

Our sparse representation removes the burden of explicitly storing anonymous variables. We now explore some variations on this theme. Prolog is capable of expressing uncertainty through variables, only for entire predicates, functors or atoms. We analyze how we may incorporate finer scale uncertainty into logical terms, specifically for arity and functors. We also propose an extension of argument indexing which permits arbitrary labels, or attributes, rather than just numerical indices. By blending these variations, applications have the ability to incorporate varying degrees of uncertainty and information into logical terms, while remaining concise and efficient.

3.1. Binding Arity

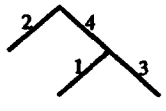
The representation we have presented does not provide a one-to-one correspondence between sparse and ordinary terms. For example, the following terms correspond to the sparse term $f - [1 - a] : f(a)$, $f(a,\_)$, $f(a,\_,\_)$, $f(a(\_),\_),\ldots$. Any sparse term has an infinite number of corresponding ordinary terms. The arity of each functor and atom is not bound, so we can always append an arbitrary number of anonymous variables as arguments of functors and atoms.

If we require the arity of terms to be bound, we must specify it explicitly, as we must do for sparse matrices. This can easily be accomplished by extending part (iii) of our definition to allow functors of the form $a/N - L$ where a is a functor, N is the arity of the functor and L is a sparse argument list. As an example, the term $f(\_, b(\_,\_), c, d(e,\_),\_)$ would be completely represented by $f/5 - [2.b/2, 3.c/0, 4.d/2 - [1.e/0]]$. The graphical representation for this term is:



3.2. Anonymous Functors

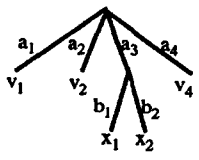
An interesting variation that we have found useful for taxonomic encoding [1], allows terms to specify only those argument positions which are occupied, but not record the functor or atom in that position. This information, presumably, would be stored elsewhere. This greatly reduces space requirements for cases when many terms are being formed from one set of data, which is indeed the case for our logical term encodings where each element of a taxonomy is assigned a term which is a subgraph of the taxonomy itself We can label the original taxonomy with term positions and use it to decode our terms. To provide functorless terms, we simply remove the functor or atom from the elements of the sparse argument list. The term $f(\_, b(\_,\_),\_, c(d,\_,e),\_)$ would thus be represented as the term [2,4-[1,3]] and graphically as:



3.3. Attribute-Value Matrices

Attribute-Value Matrices (AVMs), or Feature Structures, are a tool used in several computational linguistic systems (e.g., [2]). Some implementations of AVMs using ordinary terms require prior knowledge of all the attributes an AVM may contain in order to compile appropriate terms (e.g., [3]). A simple modification to our scheme, allowing atomic, rather than numeric, indices (for the attributes) and omitting functor names (a value is either an atom or another AVM), provides for efficient and dynamic AVMs. A predicate can be provided to access the value of an attribute, or a sequence of attributes. As an example, the sparse term $[a_1.v_1, a_2.v_2, a_3 - [b_1.x_1, b_2.x_2], a_4.v_4]$ represents the following AVM (shown in both its matrix and graphic forms):

$$\begin{bmatrix} a_1 & v_1 \\ a_2 & v_2 \\ a_3 & \begin{bmatrix} b_1 & x_1 \\ b_2 & x_2 \end{bmatrix} \\ a_4 & v_4 \end{bmatrix}$$



Attribute indexed terms have many potential applications in natural language processing and intelligent systems. As an example, consider an automatic computer configuration application (e.g., [4]) which incrementally builds components of a system, complying with some custom requirements and inherent system constraints. The overall system could be represented as an AVM in which components are named by attributes that can be successively broken down into subcomponents. Named variables could be used to ensure that the requirements and constraints are met. Backtracking over the domains of these variables could be used to find valid configurations.

3.4. Disjunctive Functors

Thus far, we have permitted two levels of certainty regarding a functor symbol: either it is unknown (i.e., it may be any atomic symbol) or it is known. Between these extremes lies a range of increasingly focussed information as to the actual functor symbol. That is, we may know that it is one of a set of possible symbols. When this set has cardinality one, we know which symbol it must be. We will name such functors *disjunctive* and represent them with a set notation. For example, the term [model.{MacSE, MacII}, memory.{1,2,4,8}] may be used to represent a computer system whose model type is either a MacSE or a MacII and with either 1, 2, 4, or 8 KB of memory.

Applications which permit and maintain uncertainty may find the flexibility offered by disjunctive functors a valuable property. Examples include computational linguistics, for maintaining the uncertainty of the referent of a pronoun, and automatic configuration.

4. CONCLUSION

We have presented a sparse approach to representing logical terms, analogous to sparse matrix representations. Applications which employ terms with many anonymous variables have the potential to benefit both in terms of storage space and unification time. We also described several variations on our representation which allow for schemes to enhance the expressiveness of terms and to incorporate finer degrees of uncertainty than that offered by variables. These variations can be combined to provide one uniform, concise and efficient sparse term representation. The straightforward nature of our proposal permits a simple implementation of the required algorithms (unification, subsumption, etc.) either in a logic language (e.g Prolog) or as an extension to a logic language (written in, e.g., C).

Our representation shares some features with the ψ-terms in LIFE [5], in particular attribute indexing and unbound arity, but it also differs in several respects. For named variables, LIFE uses more generalized coreference labels (which can specify coreference between any two locations in

the graphical representation, not just between leaves). Although our proposal implies the use of Prolog variables, we have also extended our implementation to provide both forms of coreference. Our representation also deviates from ψ-terms in the use of anonymous and disjunctive functors. Another significant difference is that our representation is intended as an enhancement to Prolog systems, not as a replacement.

REFERENCES

1. A. Fall, The foundations of taxonomic encoding, *ACM Transactions on Programming Languages and Systems* (submitted), Also available as Simon Fraser University, Technical Report SFU LCCR TR 94-20.
2. C. Pollard and I. Sag, *Information-Based Syntax and Semantics, Volume 1: Fundamentals*, CSLI Lecture Notes No. 13, Stanford, CA, (1987).
3. S. Kodric, F. Popowich, and C. Vogel, The HPSG-PL system, Version 1.2, SFU Technical Report CSS-IS TR 92-05, (1992).
4. V. Dahl, G. Sidebottom and J. Ueberla, Expert systems for automatic configuration, *International Journal of Expert Systems* **6** (4), 561–579 (1993).
5. H. Aït-Kaci and A. Podelski, Towards a meaning of life, *Journal of Logic Programming* **16** (3/4), 195 (1993).